Lessons learned from the migration to Apache Airflow
Lessons learned from the migration to Apache Airflow

Present

- Chief Architect at Skimlinks
- Trainer at Framework Training

Past

- Chief Technology Officer at A4G
- Chief Technology Officer at DML
- Data Scientist at Moore Capital
- Worked with: Orange, FCA, Kantar, OpenX and many more

Radek Maciaszek
Agenda

● Introduction
  ○ Skimlinks data pipeline
  ○ Why Airflow?

● Part I: Airflow basic concepts
  ○ Components
  ○ Features
  ○ Sample code

● Part II: How we use Airflow
  ○ Best practices
  ○ Deployment
  ○ The good, the bad and the ugly
Skimlinks Data Pipeline
Skimlinks: What we do

Publishers
60,000 publishers, including 54% of the top 100 US & UK content publishers

Merchants
50 networks giving us access to 48,500 advertisers

Metrics
81 billion page impressions p/y
Half a billion clicks
$800M Ecommerce transactions
Hundreds of TBs of data
Customer reports, data exports, predictions

Monetize product links in commerce-related content to earn publishers a share of sales.
Why Airflow?

- Scheduler (cron on steroids)
- Python: language of choice for data science
- Productivity enhancer
- Modern UI allowing to visualise executions and replay them easily with few clicks
  - Real-time logs
  - Support for retries and backfill
  - Data profiling
- Command line
- Horizontally scalable
- Code-first: use code to generate DAGs (rather than declarative YAML/XML)
- Great open source community
Data Architecture Overview
Airflow and Spark

- Much of ML development time is data engineering
  - Cleaning input data
  - ETL
  - Preparing features
  - Running series of jobs
  - Productionizing the entire data pipeline
  - Flexibility
- Apache Spark and Airflow
  - PySpark module + Airflow’s PythonOperator
  - SparkSubmitOperator (spark-submit)
  - BashOperator + spark-submit
Airflow Basic Concepts
DAG: Directed Acyclic Graph

- Data pipeline: directed graph without cycles
- Specifies tasks and dependencies between them to execute a workflow
- Complex dependencies
  - Branching, conditional execution
- Examples
  - Create a report by running an SQL and storing the results
  - Extract features for ML pipeline
  - Trigger Apache Spark job
- UI visualisation
  - Ability to re-run any task
  - Including any dependencies
Operator

- Specifies **what** gets done (think “class”)
- Operators can be
  - built-in
  - “contrib”
  - custom
- Examples
  - **PostgresOperator**: executes a query in Postgres database
  - **BashOperator**:
    
    Executes a bash command (can be combined with templates)
Task

- An instance of an operator (think “object”)
- A node in a DAG
- Specifies dynamic/static parameter values for an operator
- Jinja templating system is used to evaluate expressions at DAG run time

```python
last_op = merge_op = BigQueryOperator(
    task_id=dag_utils.default_task_id(dag.dag_id, suffix='merge'),
    retries=1,
    bql=''
    MERGE INTO `{}` `T
    USING `{}` `S
    ON `{}`
    WHEN MATCHED THEN {}
    WHEN NOT MATCHED THEN '{}'.format(
        dataset_target_tbl,
        gcs_to_bq_dataset_target_tbl,
        sql_merge_condition(merge_cond_keys),
        sql_merge_matched_clause(schema_fields, merge_cond_keys),
        sql_merge_not_matched_clause(schema_fields),
    ),
    bigquery_conn_id=dag_utils.DEFAULT_DP_GOOGLE_CONN_ID,
    use_legacy_sql=False,
    dag=dag)
```
Advanced Features

- **Hooks**
  - Interfaces to external platforms and databases
- **Connections**
  - Stored in DB, connection has an id which workflows can refer to
- **Variables**
  - Airflow allows to store arbitrary data in its database (paths, etc)
- **X-Coms**
  - Means of communication between tasks
  - A task instance object has a xcom_push and xcom_pull methods to read and write XComs
- **Sensors**
  - Pause the execution until some criterion has been met (file exists, etc)
- **Plugins**
  - Pre-packaged hooks, operators, template macros, web views, etc
  - See for example [https://github.com/airflow-plugins](https://github.com/airflow-plugins)
Sample code

```python
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime, timedelta

default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2015, 6, 1),
    'email': ['airflow@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
    # 'queue': 'bash_queue',
    # 'pool': 'backfill',
    # 'priority_weight': 10,
    # 'end_date': datetime(2016, 1, 1),
}

dag = DAG('tutorial',
           default_args=default_args,
           schedule_interval=timedelta(days=1))

# t1, t2 and t3 are examples of tasks created by instantiating operators

t1 = BashOperator(
    task_id='print_date',
    bash_command='date',
    dag=dag)

t2 = BashOperator(
    task_id='sleep',
    bash_command='sleep 5',
    retries=3,
    dag=dag)

templated_command = '
{% for i in range(5) %}
    echo "{{ ds }}"
    echo "{{ macros.ds_add(ds, 7)}}"
    echo "{{ params.my_param }}"
{% endfor %}
'

t3 = BashOperator(
    task_id='templated',
    bash_command=templated_command,
    dag=dag)

# Set t2 and t3 as the upstream tasks of t1

t2.set_upstream(t1)
t3.set_upstream(t1)
```
Airflow Best Practices
Idempotent DAGs

- Any DAG can be re-run without any side effects
  - **Repeatable** ETL process
  - Re-running DAGs is much easier
- Without idempotent data pipeline
  - Any bug can ruin the data pipeline
  - Single change can cascade across the pipeline
- Examples
  - Do not load duplicated data
  - Clean up after the steps
- More difficult to create idempotent pipeline
  - The extra effort pays off during prod issues
Best practices: Tests

- To test the task
  `airflow test <dag_name> <task_name> <date>`
- For every DAG create a Test DAG
  - Runs the same code
  - Compares the output with expected output
  - Execute integration tests during the build
  - Final step of the DAG tests the correctness of data
- Separate environments
  - Prod
  - Staging
  - Test / local Docker
Best practices: Docker and Kubernetes environments

- **Local development**
  - One liner to run a local dockerised Airflow
    ```bash
docker-compose up
    ```
- **Kubernetes cluster**
  - Horizontally scalable workers with Celery executor
  - Logs persisted in GCS
  - Ideally: fully managed cluster
- **Deployment**
  - Git-sync, rsync
  - Persistent Volume
  - Docker container
  - Airflow scans periodically files for changes
Airflow: The Good, the Bad and the Ugly

- Displaying dynamically generated DAGs might be tricky
- **DAGs dependencies**
  - Large complex DAGs vs small isolated DAGs
  - ExternalTaskSensors take up worker resources
- **We do restart the scheduler every few hours**
  - This should not be needed anymore in Airflow v1.10.3
- **Zombies**
  - Problem with killed tasks container in the k8s pod = zombie
  - Re-running things in Airflow auto-heals these problems
- **SubDAGs**
  - We embrace them, they allow encapsulation
  - Easier to visualise large pipelines with the SubDAGs
- **Productivity enhancer**